Toward Instant Gradeification

Daniel M. Zimmerman*, Joseph R. Kiniry† and Fintan Fairmichael‡

*University of Washington Tacoma, USA — dmz@acm.org
†IT University of Copenhagen, Denmark — kiniry@acm.org
‡University College Dublin, Ireland — fintan.fairmichael@ucd.ie

Abstract

Providing useful feedback to students about both the functional correctness and the internal structure of their submissions is the most labor-intensive part of teaching programming courses. The former can be automated through test scripts and other similar mechanisms; however, the latter typically requires a detailed inspection of the submitted code. This paper introduces AutoGradeMe*, a tool that automates much (but not all) of the work required to grade the internal structure of a student submission in the Java programming language. It integrates with the Eclipse IDE and multiple third-party plug-ins to provide instructors with an easy-to-use grading environment. More importantly, unlike other automatic grading tools currently in use, it gives students* continuous feedback *about their work during the development process.*

1. Introduction

Grading student submissions is the most time-consuming, labor-intensive part of teaching programming courses. Graders must provide useful feedback to students on both *external correctness*, the degree to which the submission fulfills its functional requirements, and *internal correctness*, the degree to which the submission uses language features in reasonable ways and has easily understandable code (for some definition of "easily"). Some courses may require assignments to conform to a particular coding standard, to use algorithms with time or space complexity within a given threshold, or to use sound object-oriented design principles, all of which also fall under the umbrella of internal correctness.

External correctness can be evaluated in several ways, ranging from automated testing for assignments with well-defined I/O patterns or API to manual inspection and interaction for assignments with complex student-defined graphical user interfaces. This is the more straightforward (though certainly not *easy*) part of grading: if the tests pass or the GUI reacts the way it should the submission is correct; otherwise, it is incorrect to some degree. Evaluating internal correctness is less straightforward: checking a submission's style, use of language features, object-oriented design, and complexity (both algorithmic and code) requires a detailed source code inspection rather than a brief interaction with a running program.

We present a tool called *AutoGradeMe*,[1] an Eclipse plug-in that automates much of the grading of internal correctness for object-oriented Java programming assignments. AutoGradeMe evaluates programming style, use of language features, and code complexity by using the output generated by multiple static checkers. In conjunction with the Java Modeling Language (JML) [1], the ESC/Java2 extended static checker [2], and the BONc BON [3] compiler

[1] We originally named the tool *AutoGrader*; however, we modified the name after discovering that another Java-based tool of the same name had already been released.

978-1-4577-0347-8/11/$26.00 CSEE&T 2011, Waikiki, Honolulu, HI, USA

and type checker, it can also evaluate the correctness of implementations relative to their specifications and the consistency of high-level design documentation.

AutoGradeMe does not evaluate algorithmic complexity and has limited ability to detect object-oriented design issues, so it does not eliminate the need to inspect student code; however, by automatically checking style, language feature usage, code complexity, and design consistency, it frees the grader to look at the "big picture" of a student submission rather than the minutia of syntax. More importantly, students receive *continuous feedback* from AutoGradeMe about the internal correctness of their code while their software is under construction. This continuous feedback distinguishes AutoGradeMe from other automated grading tools.

We have used AutoGradeMe in our courses at multiple institutions for over a year. During that time we have saved considerable grading effort, improved grading consistency, and virtually eliminated the element of surprise for students with respect to internal correctness grading. In the remainder of this paper we describe the operational principles and user interface of AutoGradeMe, briefly compare it to related systems, and describe our plans for future development.

2. AutoGradeMe and Eclipse

AutoGradeMe relies on the following Eclipse plug-ins to analyze student projects: *Metrics*,[2] a software metric calculator; *CheckStyle*[3] and *PMD*,[4] static checkers for Java source code; *FindBugs*,[5] a static checker for Java bytecode; *BONc*,[6] a parser and typechecker for BON; and *ESC/Java2*,[7] an extended static checker for Java.

In early programming courses, we use Metrics, CheckStyle, PMD and FindBugs to statically check student code; in more advanced courses where students must generate both informal and formal specifications for their software, we use BONc and ESC/Java2 to check specification consistency and implementation correctness. We have previously discussed our use of these plug-ins in software engineering courses [4] and as part of our development process [5].

Most of these plug-ins[8] generate Eclipse errors and warnings (hereafter, *problems*). Eclipse presents these in its problem reporter, which also reports Java compiler problems and IDE configuration problems; this can result in long lists of problems ranging from style issues (spacing, bracket placement, etc.) to code structure issues (unreachable statements, unused fields, etc.) to syntax errors. These are typically sorted by severity—errors, which indicate issues that prevent the program from compiling or running, appear at the top of the list and warnings appear below them—and can also be sorted on other attributes. Even with sorting, however, long lists of problems can be overwhelming for students and graders alike.

AutoGradeMe uses the Eclipse Platform API to examine the set of problems reported by each plug-in and distill that set into three letter grades. The first is based solely on the number of errors, the second is based solely on the number of warnings, and the third, which is used as the overall grade for the plug-in, is a weighted average of the first two. The grade scale is fully configurable, with the option to use any subset of letter grades A through G (including + and – modifiers, except for G), plus additional grades NG, meaning "no grade," and N/A, meaning "not applicable" (reported when a plug-in is turned off or its output is unavailable). Grades can

[2] http://metrics.sourceforge.net/ and http://metrics2.sourceforge.net/

[3] http://checkstyle.sourceforge.net/ and http://eclipse-cs.sourceforge.net/

[4] http://pmd.sourceforge.net/

[5] http://findbugs.sourceforge.net/

[6] http://kindsoftware.com/products/opensource/BONc/

[7] http://kindsoftware.com/products/opensource/ESCJava2/

[8] Metrics, which provides a separate Eclipse view to examine calculated project metrics, is the only exception.

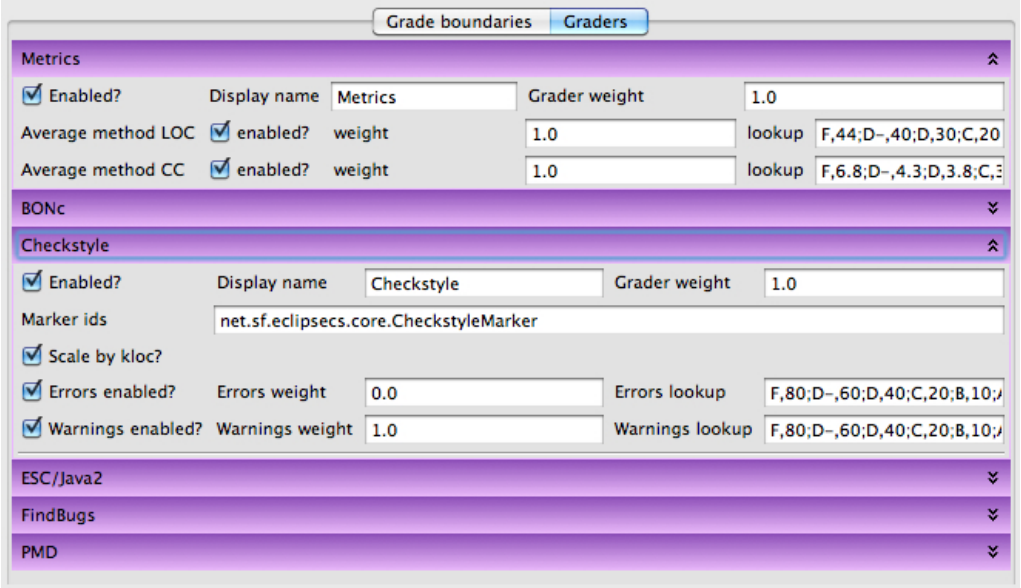

Figure 1. Configuration panel for individual plug-in graders

be individually enabled and disabled and are assigned individual colors and threshold values; for space reasons, we do not show the interface for this configuration here.

Grade computations are configured separately for each plug-in as shown in Figure 1.[9] Letter grade thresholds are set separately for error and warning grades (as well as method length and cyclomatic complexity grades); for example, the boundary between an B and a C for CheckStyle warnings in this project is 20 warnings per KLOC. Each plug-in grader has a default set of *marker IDs* that identify the problems it counts; the ability to change the marker IDs protects AutoGradeMe against future API changes by plug-in designers.

AutoGradeMe also assigns three letter grades for code metrics. The first is for the average number of lines of code in a method, for which there are generally accepted "good" ranges. The second is for the average McCabe cyclomatic complexity [6] of methods in the system. Although the utility of cyclomatic complexity has been criticized [7], we have found it useful in our courses both because it is easy for students to understand (effectively, the cyclomatic complexity of a method is the number of execution paths through the method) and because, in our experience, students (and graders) find code with lower cyclomatic complexity easier to comprehend. The third is a weighted average of the first two, either of which may be disabled.

Finally, AutoGradeMe uses a weighted average of the grades reported for all the plug-ins to compute an overall project grade. AutoGradeMe is implemented as an Eclipse *builder*, so every time a student triggers a build (e.g., by saving a modified Java source file) AutoGradeMe recalculates all the grades and updates the AutoGradeMe view. Students may choose to keep the AutoGradeMe view visible at all times to get continuous feedback on their work, to only consult the AutoGradeMe view before submission, or (at their own peril) to ignore the AutoGradeMe view completely. Instructors and graders use the AutoGradeMe view to grade the assignment. Figure 2 shows the AutoGradeMe view for student projects that received an A (above) and a B (below) on the automatically graded portions (ESC/Java2 and BONc were both disabled).

[9] In the interest of space, we have hidden some configuration options; the options for BONc, ESC/Java2, FindBugs, and PMD are essentially the same as those for CheckStyle with different marker IDs and display names.

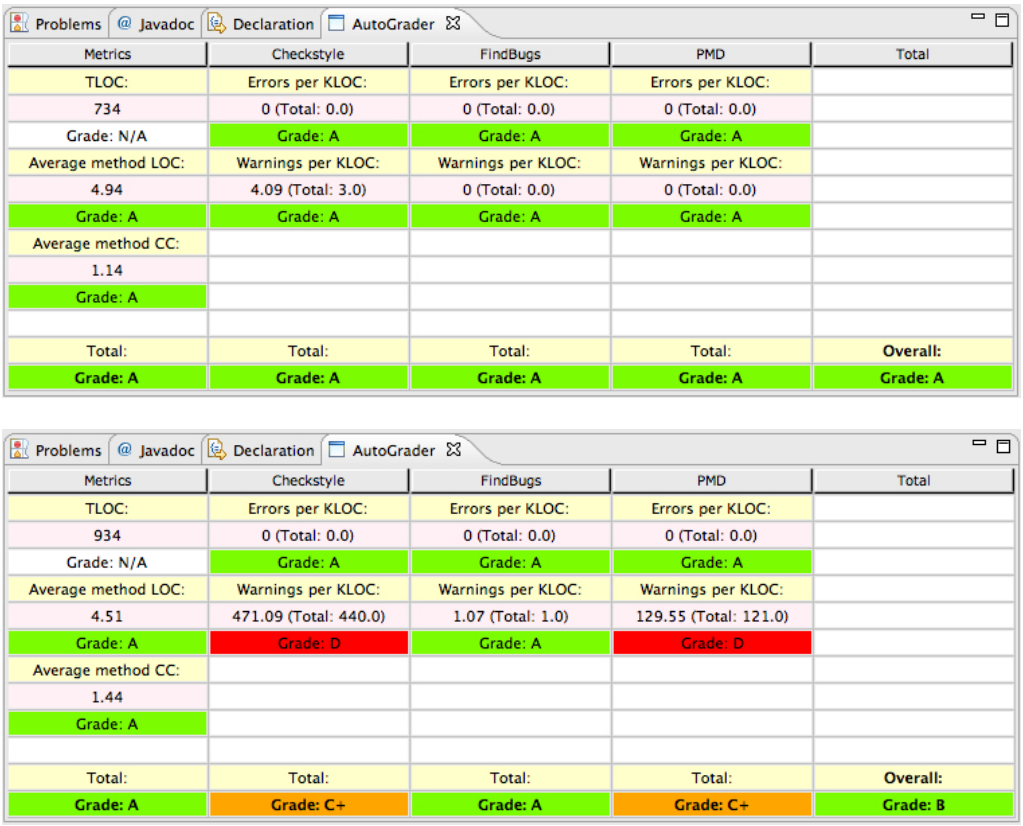

Figure 2. AutoGradeMe views for two student submissions

In order to use AutoGradeMe for an assignment, the instructor distributes an Eclipse project template; typically this takes the form of a `.zip` file that is imported into Eclipse in one step. A project template includes configurations for all the plug-ins, enabling the instructor to create customized settings for each assignment. It also includes the AutoGradeMe configuration (weights, thresholds, etc.) that will be used when grading the assignment. As long as students do not tamper with the AutoGradeMe and static checker settings, the AutoGradeMe view they see when coding will exactly match the AutoGradeMe view the instructor sees when grading.

AutoGradeMe has been publicly available for over a year. It can be easily installed from our Eclipse update site, and we maintain a publicly-visible issue tracking system for feature requests, bug reports, etc. The AutoGradeMe web page[10] contains links to both.

3. Related Work

Automatic grading has been extensively researched; the earliest published work we have found dates from 1960, when Hollingsworth [8] reported on the use of an automatic grader with a punch-card driven IBM 650. Many automatic grading frameworks have been developed since then, including recent examples such as AutoGrader [9], Web-CAT [10], and Marmoset [11].

We have insufficient space to discuss related work in detail, so we highlight the important differences between AutoGradeMe and other existing systems. The majority of existing systems deal only with external correctness aspects of student submissions, a problem that we

[10] http://kindsoftware.com/products/opensource/AutoGrader/ – the web page and Eclipse update site both reflected the original "AutoGrader" name at the time of this writing.

have not (yet, see below) addressed with AutoGradeMe. Those that do address some of the same internal correctness aspects as AutoGradeMe, such as Web-CAT and Marmoset, perform their internal correctness evaluations *after* assignment submission rather than providing the continuous feedback throughout the program development process that AutoGradeMe does. In addition, no system that we have found addresses the correctness of implementations with respect to contracts or high-level specifications, as AutoGradeMe does using ESC/Java2 and BONc respectively.

4. Future Directions

In future versions of AutoGradeMe, we intend to add the ability to automatically evaluate the outcomes of unit tests written with widely-used Java test automation frameworks. For student programs that are required to implement a specific API, a grader will then merely run a set of tests against a student Eclipse project to see how AutoGradeMe evaluates it. We also intend to add the ability to evaluate student-written unit tests on the basis of code coverage using a coverage plug-in; while high code coverage is only one characteristic of good unit tests, it is a tedious one for graders to measure by hand. These new features will allow AutoGradeMe to grade some external correctness characteristics of student submissions in addition to the internal correctness characteristics it already grades.

Another area we intend to address is extensibility. Currently, AutoGradeMe is aware of exactly the six plug-ins we have discussed. In a future version, instructors will be able to add the output of any Eclipse plug-in that generates problem reports to AutoGradeMe calculations in a uniform, straightforward way without needing to modify the AutoGradeMe code.

References

[1] L. Burdy *et al.*, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, Feb. 2005.

[2] J. R. Kiniry and D. R. Cok, "ESC/Java2: Uniting ESC/Java and JML," in *International Workshop on the Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, 2004.

[3] K. Waldén and J.-M. Nerson, *Seamless Object-Oriented Software Architecture—Analysis and Design of Reliable Systems*. Prentice–Hall, Inc., 1995.

[4] J. R. Kiniry and D. M. Zimmerman, "Secret ninja formal methods," in *15th International Symposium on Formal Methods (FM)*, 2008.

[5] D. M. Zimmerman and J. R. Kiniry, "A verification-centric software development process for Java," in *9th International Conference on Quality Software (QSIC)*, 2009.

[6] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[7] M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Software Engineering Journal*, vol. 3, no. 2, pp. 30–36, Mar. 1988.

[8] J. Hollingsworth, "Automatic graders for programming classes," *Communications of the ACM*, vol. 3, no. 10, pp. 529–529, Oct. 1960.

[9] M. T. Helmick, "Interface-based programming assignments and automatic grading of Java programs," in *12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE)*, Dundee, Scotland, 2007.

[10] S. H. Edwards, "Teaching software testing: Automatic grading meets test-first coding," in *18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2003.

[11] J. Spacco *et al.*, "Experiences with Marmoset: Designing and using an advanced submission and testing system for programming courses," in *11th Annual Conference on Innovation and Technology in Computer Science Education (ITICSE)*, 2006.